Laravel 4.2 中文文档

star001007

Published
with GitBook



目錄

Laravel 文档翻译及学习笔记

包含两部分内容：

> - 翻译官方文档laravel 4.2
> - 记录学习过程中的心得体会

介绍

从哪儿里开始

学习一个新的框架可能会比较难，但是很有趣。为了更快捷的帮助大家学习，我们尝试着创建了清晰、简明的Laravel文档。下面推荐一些您需要提前阅读的内容：

- `安装` 和 `配置`
- `路由`
- `请求` & `输入`
- `模板` & `响应`
- `控制器`

在您学习完上述内容后，你应该掌握了基本的Laravel的 `输入` / `输出` 操作。接下来，你可能阅读了 `配置数据库（configuring your databases）` ，`流畅的查询生成器（fluent query builder）` 及 `Eloquent ORM` 。或者，你想阅读 `验证和安全（authentication and security）` 来在您的应用中使用用户。

Laravel的哲学

Laravel是一个富有表达力、具有优雅语法的web框架。我们坚信开发应该是愉悦的、富有创造性体验的。Laravel尝试着通过集成大部分web项目中的功能，如 `权限` 、 `路由` 、 `会话` 和 `缓存` 等，来使得开发更轻松。

Laravel的目标是在不牺牲应用的功能的前提下，来使得程序的开发过程更加愉悦。开心的程序员创造最好的代码！为了达到这样的目的，我们尝试着结合其他框架的优势，包含其他语言如 `Ruby on Rails` 、 `ASP.NET MVC` 、 `Sinatra` 实现的框架。

Laravel是平易近人的，强大的，提供了开发大型、健壮应用的强大工具。极好的控制器反转、具有表现力的迁移系统及紧密集成的单元测试的支持，给你提供了构建任何应用的工具。

快速开始

[TOC]

安装

通过 `Laravel installer` 安装

首先，通过Composer下载 Laravel installer。

```
composer global require "laravel/installer=~1.1"
```

确保 `~/.composer/vender/bin` 目录在PATH存在，使得你在执行 **Laravel**命令时，可以找到laravel可执行文件。

一旦安装后，简单的命令 `laravel new` 命令可以在你指定的位置创建一个全新的laravel。例如， `laravel new blog` 会在当前路径下创建一个名称为**blog**的文件夹，包含了全新的laravel及所有的依赖。这种安装方法比通过 `Composer` 安装更快捷。

通过 `Composer` 安装

Laravel框架使用 Composer 来安装和依赖管理。如果你还没有安装，从 安装Composer 开始吧。 接下来你可以在终端打出下列命令来安装Laravel：

```
composer create-project laravel/laravel your-project-name --prefer-dist
```

这条命令将会在当前目录下的 `your-project-name` 文件夹下载并安装一个全新的laravel。

如果你愿意，你还可以从Githup的Laravel仓库来下载，接着在你创建的项目的根目录下运行 `composer install` 命令，这条命名会下载并安装框架的依赖。

权限（**Permissions**）

安装laravel后，你可能需要给web server在 `app/storage` 目录上写文件的权限。更多配置的详细信息看安装文档

Laravel服务器

通常情况下，你可能会用 `Apache` 或者 `Nginx` 类似的服务器来作为您的Laravel应用的服务器。如果您使用的PHP5.4版本以上，可能会想用PHP内置的开发服务器，你可以使用 **Artisan**命令 `server` ：

```
php artisan serve
```

目录结构

安装框架后，让我们熟悉下目录结构。 `app` 目录包含文件夹： `views` , `controllers` 和 `models` ，您的应用的大部分的代码都在这些目录里。你可能还想看下 `app/config` 目录及相关的配置选项。

本地开发环境

过去，配置一个本地的PHP开发环境是一件令人头疼的事情。安装合适版本的PHP、必须的扩展以及其他一些部件是一件困惑、耗时的事情。代替使用Laravel Homestead，Homestead是为Laravle和Vagrant设计的简单的虚拟机。因为Homestead vagrant box 已经提前预装好了您构建一个健壮PHP应用的所有软件，你可以在几秒钟内就创建一个虚拟的、隔离的开发环境。下面是Homestead包含的一些软件：

> - Nginx
> - PHP 5.6
> - MySQL
> - Redis
> - Memcached
> - Beanstalk

不要担心，即使“虚拟”听起来也挺复杂的。 `VirtualBox` 和 `Vagrant` ,是 `Homestead` 的两个依赖，包含适用于几乎所有开源系统的简单的、图形化的安装器 。更多内容查看Homestead 文档。

路由

开始前，让我们创建第一个路由器。在Laravel里，最简单的路由是一个闭包。打开 `app/routes.php` 文件，在文件末尾添加下列路由

```
Route:get('users', function()
{
    return 'Users!';
});
```

现在，如果你在web浏览器里输入 `/users` 路由，你将会看到响应是：Users! 太棒了，你刚刚创建一个第一个路由。

路由还可以绑定到控制器方法上。比如：

```
Route::get('users', 'UserController@getIndex');
```

这个路由告知框架请求 `/users` 路由会调用 `UserController` 控制器类的 `getIndex` 方法。了解更多的控制器路由，查看控制器文档。

创建一个视图

接下来，我们创建一个简单视图来展示user数据。视图模板都在 `app/views` 目录里，宝航了您应用的HTML。接着我们在当前目录下创建两个视图： `layout.blade.php` 和 `users.blade.php` 。首先，让我们创建 `layout.blade.php` 文件：

```
<html>
    <body>
        <h1>Laravel Quickstart</h1>
        @yield('content')
    </body>
</html>
```

接下来，我们创建 `users.blade.php`：

```
@extends('layout')

@section('content')
    Users!
@stop
```

其中的一些语法看起来相当的奇怪。这是因为我们使用了Laravel模板系统： `Blade` .Blade 非常快，因为它仅仅在模板系统上使用了极少的正则表达式来编译成纯粹的PHP代码。Blade提供了强大的功能，如模板继承，以及类似PHP控制结构的if for语法糖。查看更多信息，请看Blade文档。

现在，我们已经有了两个模板，让我们从 `/users` 的路由中返回，而不再是 `Users!` ，路由代码如下：

```
Route::get('users', function()
{
    return View::make('users');
});
```

太棒了，现在我们创建了一个继承layout的视图。接下来，让我们开始使用数据库层。

创建一个迁移

为了创建一个拥有数据的表，我们需要使用Laravel的迁移系统。迁移允许你丰富地定义对数据库的修改，并很容易将修改对您的团队进行分享。

首先，让我们配置一个数据库连接。你可以在 `app/config/database.php` 文件里配置你所有的数据库连接。 默认地，Laravel使用MySQL驱动器，你需要在数据库配置文件里提供验证信息。 如果你愿意，可以将驱动器选项改为 `sqlite` ,那么它就会使用包含在 `app/database` 目录里的SQLite数据库。

接下来，为了创建一个迁移。我们将使用 `Aitisan CLI` 。在项目的根目录，在终端执行下列代码：

```
php artisan migrate:make create_users_table
```

接下来，知道在 `app/database/migrations` 文件夹里生成的迁移文件。这个文件包含了一个类有两个方法：up和down。在up方法里，你可以进行对数据表的更改。在down方法里，你仅需要颠倒它们。

```
public function up()
{
    Schema::create('users', function($table)
```

```
    {
        $table->increments(id);
        $table->string('email')->unique();
        $table->string('name');
        $table->timestamps();
    });
}

public function down()
{
    Schema::drop('users');
}
```

最后，我们可以在终端运行 `migrate` 命令来进行迁移操作。注意：在项目的根目录里进行执行。

```
php artisan migrate
```

如果你想回滚您的迁移，使用 `migrate:rollback` 命令。现在我们已经有了一个数据表，让我们开始添加一些数据吧。

Eloquent ORM

Laravel 配备一个强大的ORM：Eloquent。如果你使用过Ruby on Rails 框架，那么将会对Eloquent非常熟悉，它也遵循了 ActiveRecord 数据库交互的风格。

首先，让我们建一个模型：一个Eloquent模型可以用来查询关联数据表，会返回数据表中指定的行。不要担心，很快会弄清楚的。模型通常存放在 `app/models` 目录下。让我们在这个目录下定义一个 `User.php` 模型如下：

```
class User extends Eloquent{}
```

注意，我们不需要告诉Eloquent我们使用哪儿个数据表。Eloquent有大量的便捷操作，其中一个就是使用模型的复数形式作为模型数据库的数据表。非常方便有木有?

使用数据库管理工具在你的users数据表里插入几行数据，我们接下来用Eloquent来获取它们并传递到我们的视图。

现在，让我们修改我们 `/users` 路由如下：

```
Route::get('users', function(){
    $users = User::all();
    return View::make('users')->with('users', $users);
});
```

我们一行行的看下上面的路由。首先，模型 `Users` 的 `all` 方法会获取数据表users的所有的行。接下里，我们通过 `with` 方法来传递这些结果行到View视图。这个 `with` 方法接收两个参数：key和value，以在视图中使用。

太棒了，现在我们已经准备好在我们的视图中展示`users`表的数据了。

展示数据

现在让我们使用上面的`$users`变量，我们可以像下面这么做：

```
@extends('layout')

@section('content')
    @foreach($users as $user)
        <p>{{ $user->name }}</p>
    @endforeach
@stop
```

你可能想问，我们的`echo`在哪儿里呢？其实 当我们使用`Blade`时，你输出的数据需要用两个大括号来引起来。这个很简单，现在我们可以再浏览器中输入`/users`路由，那么将会看到结果中会有我们的users信息。

这仅仅是个开始。在这个教程里，你只看到了Laravel的最基本的东西，但是还有很多激动人心的事情等着我们去发掘。阅读 `Eloquent` 和 `Blade` 的详细文档，会让你发现它的强大特性。当然，你也可能会对`队列：Queues` 和 `单元测试：Unit Testing` 更感兴趣。 话说回来，如果你想通过 Ioc容器来更改你的架构，这一切都由你来做主。

部署你的应用

Laravel的其中一个目标就是让PHP应用开发从下载到部署都是简单快乐的， Laravel Forge 就可以让您的Laravel应用很简单的部署到超级快的服务器上面。 Forge 可以配置、提供DigitalOcean/Linode/Amazon EC2 服务器。如Homestead一样，所有最新的软件如：Nginx/PHP 5.6/MySQL/Postgres/Redis/Memcached 等都已经被安装好了。Forge的快速部署**[Quick Deploy]**甚至可以在你每次给Github或Bitbucket提交修改时，同时将你的代码就行同步更新。

除了这些，Forge可以帮助你配置`队列进程:queue workers`/`SSL`/`Cron jobs`/`sub-domains`等，更多信息，请查看Forge website

版本说明

Laravel 4.2版本

你可以通过命令：`php artisan changes` 来查看这次版本的所有变化，或者可以通过Githup文件。这些说明涵盖了所有的主要改进及变化。

> 注意：在4.2版本里，许多小的bug修复及改进都体现到了4.1版本中。所以，你可以同时看下4.1版本的更改列表。

需要**PHP5.4+**版本以上支持

Laravel4.2需要PHP5.4或者更改的版本。为了使用PHP的新特性如 `traits`，以提供更具表现力的接口如：laravel Cashier。PHP5.4相比5.3同样带来了速度跟性能上的提升。

Laravel Forge

Laravel Forge是全新的基于web的应用，使得在云上，如Linode, DigitalOcean, Rackspace, and Amazon EC2 创建和管理PHP服务器变得非常简单。支持自动化的配置，SSH密钥的访问，后台任务的自动化，NewRelic & Papertrail的服务器监控，“更改即部署”，Laravel进程队列配置等等。Forge提供所有Laravel应用运行的最简单、最经济实惠的办法。

Laravel4.2的配置文件 `app/config/databases.php` 默认使用了Forge,以更快捷的来进行部署。

更多关于Laravel Forge的信息，查看 Forge官方网站

Laravel Homestead

Laravel Homestead 是官方的vagrant环境用来部署健壮的Laravel和PHP应用。The vast majority of the boxes' provisioning needs are handled before the box is packaged for distribution, allowing the box to boot extremely quickly. Homestead includes Nginx 1.6, PHP 5.6, MySQL, Postgres, Redis, Memcached, Beanstalk, Node, Gulp, Grunt, & Bower. Homestead includes a simple Homestead.yaml configuration file for managing multiple Laravel applications on a single box.

The default Laravel 4.2 installation now includes an app/config/local/database.php configuration file that is configured to use the Homestead database out of the box, making Laravel initial installation and configuration more convenient.

The official documentation has also been updated to include Homestead documentation.

Laravel Cashier

Laravel Cashier is a simple, expressive library for managing subscription billing with Stripe. With the introduction of Laravel 4.2, we are including Cashier documentation along with the main Laravel documentation, though installation of the component itself is still optional. This release of Cashier brings numerous bug fixes, multi-currency support, and compatibility with the latest Stripe API.

Daeamon Queue Workers

The Artisan queue:work command now supports a --daemon option to start a worker in "daemon mode", meaning the worker will continue to process jobs without ever re-booting the framework. This results in a significant reduction in CPU usage at the cost of a slightly more complex application deployment process.

More information about daemon queue workers can be found in the queue documentation.

邮件API驱动

Laravel 4.2 introduces new Mailgun and Mandrill API drivers for the Mail functions. For many applications, this provides a faster and more reliable method of sending e-mails than the SMTP options. The new drivers utilize the Guzzle 4 HTTP library.

软删除接口

A much cleaner architecture for "soft deletes" and other "global scopes" has been introduced via PHP 5.4 traits. This new architecture allows for the easier construction of similar global traits, and a cleaner separation of concerns within the framework itself.

More information on the new SoftDeletingTrait may be found in the Eloquent documentation.

方便的权限 & Remindable Traits

The default Laravel 4.2 installation now uses simple traits for including the needed properties for the authentication and password reminder user interfaces. This provides a much cleaner default User model file out of the box.

简单分页

A new simplePaginate method was added to the query and Eloquent builder which allows for more efficient queries when using simple "Next" and "Previous" links in your pagination view.

迁移确认

In production, destructive migration operations will now ask for confirmation. Commands may be forced to run without any prompts using the --force command.

Laravel 4.1版本

全部变化

新的SSH组件

升级说明

贡献指南

贡献指南

安装

安装composer

Laravel使用composer来管理依赖。首先，下载一份 `composer.phar` ,安装后，你可以在当前项目目录中使用或者移动到 `/usr/local/bin` 作为全局命令来使用。在windows里，你可以使用composer安装器。

安装Laravel

通过Laravel安装器

首先，通过composer下载Laravel安装器：

```
composer global require "laravel/installer=~1.1"
```

确保 `~/.composer/vendor/bin` 在你的PATH环境变量里，使Laravel执行时可以找到。

一旦安装成功后，简单的使用命令： `laravel new` 就可以在你指定的目录创建一个全新的laravel。比如， `laravel new blog` 会创建一个目录为blog的并包含所有依赖的laravel。这种安装相比较通过composer安装时非常快捷的。

通过composer create-project 命令

使用composer的 `create-project` 命令：

```
composer create-project laravel/laravel --prefer-dist
```

直接下载

安装composer后，下载最新版本的laravel框架并解压。接下来，在你laravel应用的根目录，执行 `php composer.phar install` 或者 `composer install` 命令来安装框架的依赖。为了顺利完成安装，这个过程还需要你有安装git。

服务器环境的要求

laravel框架需要如下的系统要求：

> - PHP版本>=5.4
> - PHP的Mcrypt扩展已安装

PHP5.5以上的版本，一些操作系统可能还需要你安装JSON扩展。当你使用Ubuntu时，你可以执行 `apt-get install php5-json` 来完成。

配置

完成安装Laravel后的第一件事情就是将你的应用key设置为一个随机的字串。如果你通过composer来安装的laravel，这个key可能已经通过 `key:generate` 命令被设置好了。通常情况下，这个字串的长度是32个字符。它是在 `app.php` 配置文件中可以进行配置。如果应用的key没有被设置，那么你的应用的用户session及其他的编码的数据的安全性将会受到威胁。

Laravel几乎不需要怎么配置，就可以欢快的开始编码啦。然而，你可能还想再看下 `app/config/app.php` 文件及相关文档。它包含了一些选项比如： `timezone` 和 `locale` 可能根据您的应用需要进行更改。

当laravel安装后，你同样也需要配置下本地环境，在本地进行测试时候来获取更详细的错误信息。默认地，在生产环境的配置文件里详细的错误报告是被禁用的。

> 注意：在生产环境下，千万别把 `app.debug` 设置为true.

权限

laravel需要一些权限配置， `app/storage` 目录需要web服务器的写操作的权限。

路径

框架的几个目录路径是可以配置的。想改变这些目录的位置，请查看文件 `bootstrap/paths.php` 。

优美的URLs

Apache

框架的 `public/.htaccess` 文件可以设置url去除index.php。如果你使用的web服务器是Apache，确保你开启了 `mod_rewrite` 模块。 如果这个 `.htacess` 文件在Apache里不起作用，试下如下代码：

```
Options +FollowSymLinks
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.php [L]
```

Nginx

对于Nginx，使用下列的设置：

```
location / {
    try_files $uri $uri/ /index.php?$query_string;
}
```

配置

介绍

Laravel框架的所有配置信息都在 `app/config` 目录里。每个选项都有详细的文档说明。你可以先熟悉一下。

有时你可能想在程序执行时获取这些配置的值，使用 `Config` 类。

获取某个配置的值

```
Config::get('app.timezone');
```

如果配置选项不存在时候，你可以指定一个默认值，如下：

```
$timezone = Config::get('app.timezone', 'UTC');
```

设置某个配置的值

可能你已经注意到了， `.` 分隔符用来获取多个文件中的值。

```
Config::set('database.default', 'sqlite');
```

不过，它只在当前请求中有效。

环境配置

对于不同的运行环境，一般会有不同的配置选项。比如，线上线下环境你可能使用不同的缓存驱动器。通过环境配置选项可以很容易的来实现。

只需要在 `config` 目录里创建一个新的文件夹，命名为与环境相关的名称。接下来，创建你想覆盖和配置一些选项的配置文件。比如你想覆盖线下环境里的缓存驱动器，你可以在 `app/config/local` 里创建 `cache.php` 文件，内容如下：

```
<?php

    return array(
        'driver' => 'file',
    );
```

> 注意：不要使用 `testing` 作为环境的名称。因为它是单元测试的名称。

你会发现你并没有指定配置文件里所有的选项，而只是你想覆盖的一些选项。因为它会继承这些基本的配置文件。

接下里，你要指定框架使用哪儿个环境。默认地是 `production`，即生产环境。不过，你也可以在 `bootstrap/start.php` 文件里指定使用环境。这个文件里，你会发现 `$app->detectEnvironment` 调用，这个数组会决定你使用哪儿个环境。你也可以根据需要给这个数组添加其他环境和机器名称。

```
<?php
$env = $app->detectEnvironment(array(
    'local' =>  array('your-machine-name'),
));
```

上面例子中，`local` 是你的环境的名称，`your-machine-name` 是你的服务器的主机名。在linux或者Mac系统树上，你也可以通过终端命令：`hostname` 来指定你的主机名。

如果你想要更灵活的检测环境。你可以给 `detectEnvironment` 方法传递一个闭包，可以根据你的意愿来是指定环境。

```
$env = $app->detectEnviroment(function(){
    return $_SERVER['MY_LARAVEL_ENV'];
});
```

获取当前应用的环境

使用 `environment` 方法如下：

```
$environment = App::environment();
```

也可以给 `environment` 方法传递一个环境参数来确定是否是当前环境。

```
if (App::environment('local'))
{
    // The environment is local
}

if (App::environment('local', 'staging'))
{
    // The environment is either local OR staging...
}
```

Provider 配置

在使用环境配置时候，若想给在基础的 `app` 配置文件中追加环境 `service providers` 时候，直接追加会覆盖基础的 `app providers`。若想强制的追加，你需要在环境的app配置文件里使用 `app_config` 辅助方法如下：

```
'providers' => append_config(array(
    'LocalOnlyServiceProvider',
))
```

敏感数据保护的配置

对于生产环境来说，我们强烈建议您将所有的敏感信息的配置都放到配置文件之外。比如数据库密码，api密钥，及编码密钥应该尽可能的移到配置文件之外。 那么，我们应该放到哪儿里呢？ 幸运地是，laravel给我们提供了一个简单的解决办法，就是使用"点"文件(dot files)。

首先，配置你的应用来识别你的机器作为本地环境的机器。接下来，在你项目的根目录创建 `.env.local.php` 文件，就在 `composer.json` 文件同一目录里。这个文件返回一个键值对的数组，如所有的laravel配置文件一样：

```php
<?php

    return array(
        'TEST_STRIPE_KEY' => 'super-secret-sauce',
    );
```

这个文件的所有键值对都可以通过$_ENV 和$_SERVER 超全局变量访问。现在你可以在你的配置文件里访问这些超全局变量:

```
'key' => $_ENV['TEST_STRIPE_KEY']
```

记住将 `.env.local.php` 文件添加到 `。gitignore` 文件里，以方便团队开发时，使用各自的配置文件同时保护自己的敏感数据。

现在，你也可以在你的生产环境的项目根目录中创建 `.env.php` 文件，包含对应的值。同样滴，在版本控制器中忽略它。

```
* 注意：你可以为每个环境创建一个配置文件。在开发环境创建`.env.development.php`,在生成环境中创建`.env.
```

维护模式

当你的项目是在维护模式下，一个自定义的视图可以在应用中所有的路由里使用。当你更新或者进行维护时，可以很方便的禁用您的应用程序。在 `app/start/global.php` 文件中已经提供了 `App::down` 的方法，它的响应会在你启用维护模式后发给所有的用户。

启用维护模式，执行下列命令：

```
php artisan down
```

禁用维护模式，执行下列命令：

```
php artisan up
```

自定义视图的使用，需要在 `app/start/global.php` 里设置下：

```
App::down(function()
{
    return Response::view('maintenance', array(), 503);
});
```

如果down的匿名函数返回 `null` ,那么维护模式将会被所有请求所忽略。

维护模式&队列

如果你的应用在维护模式中，那么队列任务将不会被执行。一旦维护模式结束后，队列任务将开始正常执行。

Homestead

介绍

laravle 始终贯彻让php的开发体验更愉悦的理念，包括本地开发环境。 `Vagrant` 提供了一个简易、优雅的方法去管理和提供虚拟主机。

laravel Homestead 是一个官方的，预装Vagrant的#盒子#， 它提供了一个非常棒的环境不需要你在本机上安装php/hhvm/web服务器/其他服务软件。不用担心会弄乱你的操作系统，vagrant盒子是可完全开放的。如果过程中出错了，你可以在几分钟内在重建一个盒子。

homestead可以在任何的windows/mac/linux系统上运行，包括了nginx服务器，php5.6，MySQL，Postgres，Redis，Memcached以及其他一些好东西，让你开发你伟大的laravel应用。

```
注意：如果你在使用Windows，你可能需要开发hardware virtualization(vt-x)。它一般可以通过BIOS来开启。
```

Homestead现在用vagrant1.6来构建和测试。

包含的软件

```
Ubuntu 14.04
PHP 5.6
HHVM
Nginx
MySQL
Postgres
Node（集成了Bower，Grunt和Gulp）
Redis
Memcached
Beanstalkd
Laravel Envoy
Fabric + HipChat Extension
```

安装和设置

安装VirtualBox 和Vagrant

在运行你的Homestead环境之前，你需要安装VirtualBox和Vagrant。这些软件包提供了所有流行操作系统的可视化的安装器。

添加Vagrant盒子

VirtualBox和Vagrant安装完毕后，就可以使用下列的命令来添加 `laravel/homestead` 盒子到你的Vagrant安装。

```
vagrant box add laravel/homestead
```

需要几分钟来下载这个盒子，根据你的网络环境花费时间不同。 如果失败了，可能是你的vagrant版本问题，它需要盒子的url地址，你需要如下操作：

vagrant box add laravel/homestead https://atlas.hashicorp.com/laravel/boxes/homestead

安装Homestead

通过Comoposer和PHP工具

一旦盒子已经添加到你的Vagrant安装，你就可以通过Composer的 `global` 命令来安装Homestead 命令行工具。

```
composer global require "laravel/homestead=~2.0"
```

确保 `~/.composer/vendor/bin` 目录在你的环境变量，当执行 `homestead` 时，才能找到。

安装完Homestead命令行工具后，运行 `init` 命令来创建 `Homestead.yaml` 配置文件：

```
homestead init
```

会在 `~/.homestead` 目录下生成文件 `Homestead.yaml` ，Mac或者Linux系统，你可以通过命令 `homestead edit` 来编辑这个文件。

```
homestead edit
```

通过Git手动安装 (没有本地PHP)

相应地，如果你不想在本机上安装PHP，你可以通过克隆版本来手动安装Homestead。考虑到克隆一个版本到所有laravel项目所在的目录作为 `Homestead` 目录，这样Homestead盒子将作为主机来服务所有的laravel项目：

```
git clone https://github.com/laravel/homestead.git Homestead
```

安装完毕后，运行 `bash init.sh` 命令来创建 `Homestead.yaml` 配置文件：

```
bash init.sh
```

创建的 `Homestead.yaml` 文件在 `~/.homestead` 目录下。

设置你的SSH Key

接下里，你需要编辑 `Homestead.yaml` 文件。这个文件里，你可以配置你的公钥的路径，以及主机跟Homestead虚拟机共享目录。

若没有SSH Key，Mac或者Linux系统，你可以通过下列命令生产SSH key 对：

```
ssh-keygen -t rsa -C "you@hoemstead"
```

若是Windows，你可以按照Git 通过使用Git自带的 `Git Bash` shell来执行上面的命令。若没有，你也可以用 `PuTTY` 和 `PuTTYgen` 。

创建好SSH Key之后，指定配置文件 `Homestead.yaml` 中 `authorize` 属性为key的路径。

配置共享文件夹

`Homestead.yaml` 文件中的属性 `folders` 列出所有你想与Homestead环境共享的文件夹。文件夹中的文件发生改变，会在本机跟Homestead环境中保持同步，你可以根据需要来指定任意数量的文件夹。

配置你的Nginx站点

如果你之前对Nginx不熟悉，没关系。属性 `sites` 使得你很容易的把一个域名跟Homestead环境中的文件夹映射好。 `Homestead.yaml` 文件中包含了一个简单站点的配置。当然，你可以根据需要来给你的Homestead环境添加任意站点。Homestead服务于你运行每个laravel项目，是一个非常简洁的虚拟化环境。

你可以让Homestead站点支持 `HHVM` ，只需要设置 `hhvm` 选项的值为 `true`：

```
sites:
    - map: homestead.app
      to: /home/vagrant/Code/laravel/public
      hhvm: true
```

Bash别名

需要给Homestead添加bash别名的话，只需要添加到 `~/.homestead` 目录的 `aliases` 文件中。

运行Vagrant盒子

根据你的需要编辑过 `Homestead.yaml` 后，运行 `homestead up` 命令。如果你是手动安装的homestead，而不是使用的PHP `homestead` 工具，在git克隆的homestead的文件夹里，运行 `vagrant up` 命令。

vagrant会运行虚拟机，并且自动配置你的共享文件夹和Nginx站点。想删除的话，使用 `homestead destroy` 命令。查看所有可用的homestead命令，运行 `homestead list` 。

不要忘记在 `host` 文件里添加nginx站点的域名！host文件会讲你本地的域名指向安装的homestead环境。Mac和linux，这个文件在 `/etc/hosts` 。对于windows，在 `C:\Windows\System32\drivers\etc\hosts` 。你需要添加类似下面的记录：

```
192.168.10.10 homestead.app
```

确保你的ip地址是你在 `Homestead.yaml` 中配置的地址。 `hosts` 文件中添加后，你可以通过浏览器访问这个站点。

```
http://homestead.app
```

下面会学习如何连接数据库！

日常使用

通过ssh连接homestead

若想通过ssh连接homestead环境，只需要在终端命令行中输入 `homestead ssh` 或者 `vagrant ssh` 就可以登录。

连接数据库

`homestead` 数据库配置成mysql或者postgres。为了更方便，laravel本地的数据配置被设置成默认使用这个数据库。 需要连接的话，可以通过你主机的navicat或者sequel pro连接，连接主机是 `127.0.0.1` ， 端口是 `33060` (mysql)， `54320` (postgres)。用户名密码是 `homestead/secret` 。

```
注意：你只能用这些非标准的短裤来连接数据库。默认的3306/5432端口是laravel数据库配置文件中指定需要使用的,
```

添加其他站点

homestead环境安装好并且运行后，你可以根据需要添加其他站点，运行任意个laravel在一个虚拟的homestead环境中。首先：在 `homestead.yaml` 文件中添加站点，接着，运行 `vagrant provision` 。

同时，你也可以使用homestead环境中可用的 `serve` 脚本。想使用 `serve` 脚本，ssh登入homestead环境，接着运行下面命令：

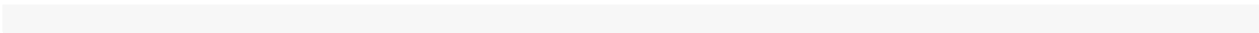
```
serve domain.app /home/vagrant/Code/path/to/public/directory

注意：运行`serve`命令后，不要忘记在`hosts`中添加域名。
```

端口

下面端口是指向你的homestead环境：

```
ssh:2222 -> forwards to 22
http:8000 -> forwards to 80
mysql:33060 -> forwards to 3306
postgre:54320 -> forwards to 5432
```



请求的生命周期

概述

所谓工欲善其事必先利其器。网站应用开发也是如此，理解工具的机理对你很有帮助，你会更有自信和更随心的使用它们。本文档的目的就是让你全面的、深刻的理解laravel框架的工作机理。随着你的理解加深，揭开它的迷纱，就可以更自如的创建应用。我们从 "启动文件"和 应用事件 来开始我们的了解。

刚开始你可能不是理解所有的东西，没关系，不要灰心。你只需要领会基本的东西，随着你阅读其他章节你会慢慢的理解它。

请求的生命周期

所有的请求都是从 `public/index.php` 入口的。当我们使用Apache时， `.htaccess` 文件会帮助我们讲所有的请求指向index.php文件。从这里开始，laravel开始处理请求并作出相应。为此，我们从laravel的启动过程开始。

到目前为止，我们了解到laravel启动过程的最重要的是服务提供商（service providers）。在 `/app/config/app.php` 配置文件里，提供了一系列的服务提供商，还有一个providers数组。这些提供商作为larvel启动机制的主要部分。再回到 `index.php` 文件，一个请求进入 `index.php` 文件后，会加载 `bootstrap/start.php` 文件，这个文件创建了laravel应用对象，同时是Ioc容器。

创建应用对象后，需要设置几个项目路径和环境检测。接着颞部的启动脚本会被执行，这个文件作用于laravel内部代码，根据你的配置文件的设置来进行设置，比如 `timezone` 、 `error_reporting` 等。更重要的，不是这些选项的设置，而是对应用中配置的服务器提供商(`service providers`)进行注册。

简单的服务器提供商只有一个方法： `register` 。应用对象通过自己的 `register` 方法注册 `service providers` 时候， `service providers` 的 `register` 方法才会被调用。一般来讲，每个 `service provider` 绑定一个或者多个闭包到容器里。在容器里允许你访问这些绑定的服务商。比如， `QueueServiceProvider` 服务器服务注册了一些闭包，来解决涉及类的一些队列。当然，服务器提供商也可以再启动任务里使用而不仅仅是注册到容器里。服务器提供商可以注册 为事件监听、视图组件、 `Artisan` 命令等。

当所有的服务提供商被注册后， `app/start` 文件会被加载。最后， `app/routes.php` 文件会被调用。一旦 `routes.php` 文件被加载后，请求对象会被转发给应用以方便它转发给其他的路由。

让我们总结一下：

> 1. 请求进入 `public/index.php` 文件
> 2. `bootstrap/start.php` 文件创建一个应用对象并进行环境监测
> 3. 内部 `framework/start.php` 文件配置选项并加载服务器提供商
> 4. 应用 `app/start` 文件会被加载
> 5. 应用 `app/routes.php` 文件会被加载
> 6. 请求对象转发给应用，并返回响应对象
> 7. 响应对象发送到终端

现在，你对于一个进入laravel的请求如何被处理有个大致的了解。接下来，我们看下启动文件。

启动文件

你的应用的启动在 `app/start.php` 里。默认地，会包含三个文件：`global.php` , `local.php` , `artisan.php` 。对于 `artisan.php` 更多细节，查看Artisan Comman Line。

`global.php` 文件默认包含了一些基本选项，比如：`Logger` 的注册、引入 `app/filters.php` 文件。你也可以根据需要自己添加，它会在每个请求中都引入。 `local.php` 文件只有在 `local` 环境里被调用。关于环境更多信息，请看配置文档。

当然，如果你有其他的环境，你也可以创建响应的启动文件，它会在对应环境中自动引入。比如，你配置文件 `bootstrap/start.php` 设置为 `development` 环境，那么这个应用当前环境下的所有请求都会引入 `app/start/development.php` 文件。

启动文件里放什么内容

启动文件就是用来存放启动程序的相关代码，比如，你可以注册一个视图组件、配置你的日志选项及一些PHP的设置等。所有的一切，你都可以自由设置。但是，对于大型的项目来说，太多的启动代码会使得启动文件都别混乱，这时候你可以考虑将一些代码移到 服务器提供商。

应用事件

注册应用事件

捏可以在请求的前后的过程中，注册 `before` 、 `after` 、 `finish` 和 `shutdown` 应用事件：

```
App::before(function($request)
{
    //
});

App::after(function($request, $response)
{
    //
});
```

这些事件的监听会在应用每个请求的 `before` 和 `after` 被调用，它可以用来过滤或者修改请求的响应。你可以注册他们到启动文件中或者一个服务器提供商里。

你也可以注册监听到 `matched` 事件上，当一个请求匹配到一个未执行的路由上，则会被调用：

```
Route::matched(function($route, $request)
{
    //
});
```

`finish` 事件会在响应被发送到终端后才被调用，它可以用来处理应用最后的任务。 `shutdown` 事件则在所有的 `finish` 事件处理完所有事件后会被调用，它是脚步执行终止之前最后的处理机会。大多时候，你不需要用到这些事件。

路由

基本路由

应用大部分的路由都在 `app/routes.php` 文件里定义。最简单的路由有一个url和必包回调组成。

基本GET路由

```
Route::get('/', function()
{
    return 'Hello World';
});
```

基本的POST路由

```
Route::post('foo/bar', function()
{
    return 'Hello World';
});
```

注册一个路由到多种HTTP方式

```
Route::match(array('GET', 'POST'), '/', function()
{
    return 'Hello World';
});
```

注册一个路由到所有的HTTP方法

```
Route::any('foo', function()
{
    return 'Hello World';
});
```

强制路由必须是HTTPS方式

```
Route::get('foo', array('https', function()
{
    return 'Must be over HTTPS';
}));
```

如果你想获取路由生成的url，可以通过 `URL::to` 方法来获取：

```
$url = URL::to('foo');
```

路由参数

```
Route::get('user/{id}', function($id)
{
    return 'User '.$id;
});
```

可选路由参数

```
Route::get('user/{name?}', function($name = null)
{
    return $name;
});
```

可选路由参数，有默认值

```
Route::get('user/{name?}', function($name = 'John')
{
    return $name;
});
```

正则表达式，路由约束（通过where）

```
Route::get('user/{name}', function($name)
{
    //
})
->where('name', '[A-Za-z]+');

Route::get('user/{id}', function($id)
{
    //
})
->where('id', '[0-9]+');
```

where数组，路由约束

```
Route::get('user/{id}/{name}', function($id, $name)
{
    //
})
->where(array('id' => '[0-9]+', 'name' => '[a-z]+'))
```

定义全局模式

如果你想对某一个路由参数始终用某一个特定表达式来约束，那么你可以使用 `pattern` 方法：

```
Route::pattern('id', '[0-9]+');

Route::get('user/{id}', function($id)
{
    // Only called if {id} is numeric.
});
```

获取路由参数的值

如果想在路由外面使用路由参数，你可以通过 `Route::input` 方法来获取：

```
Route::filter('foo', function()
{
    if (Route::input('id') == 1)
    {
        //
    }
});
```

路由过滤器

路由过滤器可以通过限制路由很容易地帮你实现网站的验证。laravel框架有 `auth` 过滤器， `auth.basic` 过滤器， `gust` 过滤器和 `csrf` 过滤器，它们都在 `app/filters` 文件夹下。

定义一个路由过滤器

```
Route::filter('old', function()
{
    if (Input::get('age') < 200)
    {
        return Redirect::to('home');
    }
});
```

如果这个过滤器返回一个响应，那么会作为请求的响应，这个路由器将不会被执行。这个路由的 `after` 过滤器也会取消执行。

给路由绑定过滤器

```
Route::get('user', array('before' => 'old', function()
{
    return 'You are over 200 years old!';
}));
```

给控制器绑定过滤器

```
Route::get('user', array('before' => 'old', 'uses' => 'UserController@showProfile'));
```

一个路由绑定多个过滤器

```
Route::get('user', array('before' => 'auth|old', function()
{
    return 'You are authenticated and over 200 years old!';
}));
```

一个路由绑定多个过滤器（通过**array**）

```
Route::get('user', array('before' => array('auth', 'old'), function()
{
    return 'You are authenticated and over 200 years old!';
}));
```

指定路由参数

```
Route::filter('age', function($route, $request, $value)
{
    //
});

Route::get('user', array('before' => 'age:200', function()
{
    return 'Hello World';
}));
```

`after` 过滤器会接收 `$response` 作为郭略器的第三个参数：

```
Route::filter('log', function($route, $request, $response)
{
    //
});
```

基于模式的过滤器（**when**方法）

你可以根据url，对某一系列的路由指定一个过滤器：

```
Route::filter('admin', function()
{
    //
});

Route::when('admin/*', 'admin');
```

上面例子中，对于所有以 `admin/` 开头的路由都使用 `admin` 过滤器。 `*` 符号会作为通配符，会匹配所有组合的字符。

你还可以通过HTTP方式来约束：

```
Route::when('admin/*', 'admin', array('post'));
```

过滤器类

更高级的过滤器是用一个类替代闭包函数。过滤器类由应用的 `Ioc` 容器来解析，那么你可以在过滤中利用依赖注入，极大方便测试。

注册一个类过滤器

```
Route::filter('foo', 'FooFilter');
```

默认地，会调用 `FooFilter` 类的 `filter` 方法：

```
class FooFilter {

    public function filter()
    {
        // Filter logic...
    }

}
```

如果你想调用其他方法，指定它就行：

```
Route::filter('foo', 'FooFilter@foo');
```

路由命名

命名路由在路由生成跳转和urls时候，指向路由更方便。你可以如下给路由命名：

```
Route::get('user/profile', array('as' => 'profile', function()
{
    //
}));
```

你可以给控制器的操作来命名：

```
Route::get('user/profile', array('as' => 'profile', 'uses' => 'UserController@showProfile
```

那么你就可以使用如下：

```
$url = URL::route('profile');

$redirect = Redirect::route('profile');
```

你可以通过 `currentRouteName` 方法来获取当前路由的名称

```
$name = Route::currentRouteName();
```

路由组

当你需要给一组路由指定过滤器时，而不是每个路由都去指定时候，可以使用路由组：

```
Route::group(array('before' => 'auth'), function()
{
    Route::get('/', function()
    {
        // Has Auth Filter
    });

    Route::get('user/profile', function()
    {
        // Has Auth Filter
    });
});
```

你也可以在路由组的数组里使用 `namespace` 参数来限定这个组的控制器，保证它们在同一个命名空间内。

```
Route::group(array('namespace' => 'Admin'), function()
{
    //
});
```

子域名路由

laravel路由也可以处理通配符子域名，可以传递通配符参数。

注册子域名路由

```
Route::group(array('domain' => '{account}.myapp.com'), function()
{

    Route::get('user/{id}', function($account, $id)
    {
        //
    });
```

```
});
```

路由前缀

组路由可以通过组array的属性选项 `prefix` 使用前缀:

```
Route::group(array('prefix' => 'admin'), function()
{

    Route::get('user', function()
    {
        //
    });

});
```

路由模型绑定

模型绑定可以方便的将模型实例注入到路由中。比如，你可以注入一个用户ID对应的模型实例，而不单单是一个用户ID 。首先， `Route::model` 方法指定一个给定参数的模型。

绑定一个参数到模型

```
Route::model('user', 'User');
```

接下来，定义一个含有 `{user}` 参数的路由如下：

```
Route::get('profile/{user}', function(User $user)
{
    //
});
```

因为我们之前已经将 `{user}` 参数绑定到了 `User` 模型，所以 `User` 实例将会注册到这个路由。也就是说，一个请求 `profile/1` 将会注册一个ID是1的用户实例。

> - 注意：如果匹配的模型实例在数据库中没有找到，那么会抛出一个404错误。

如果你想指定 `not found` 行为，你可以给 `model` 方法的第三个参数的传递一个 闭包函数：

```
Route::model('user', 'User', function()
{
    throw new NotFoundHttpException;
});
```

如果你想自己实现路由参数的解析，你可以使用 `Route::bind` 方法：

```
Route::bind('user', function($value, $route)
{
    return User::where('name', $value)->first();
});
```

抛出404错误

路由里，有两种方法可以手动触发404错误。第一种是：

```
App::abort(404);
```

第二种是抛出异常：`Symfony\Component\HttpKernel\Exception\NotFoundHttpException` 的实例。

更多关于404异常信息及自定义错误响应在errors里说明。

路由到控制器

laravel不仅可以路由到闭包，也可以路由到控制器类，甚至到创建的 `resource` 控制器。

更多信息，看控制器。

请求和输入

基本输入

只需要简单几个方法就可以获取用户所有输入。你不需要考虑用户HTTP请求方法，不管哪儿种请求方式，都是一样的获取方式。

获取输入的值

```
$name = Input::get('name');
```

如果输入值不存在，设定默认值

```
$name = Input::get('name', 'Sally');
```

判断输入值是否存在

```
if (Input::has('name'))
{
    //
}
```

获取请求的所有输入

```
$input = Input::all();
```

获取请求的部分输入

```
$input = Input::only('username', 'password');   // 只有

$input = Input::except('credit_card');  //除了
```

若输入是数组时，你可以通过 `.` 符号来获取：

```
$input = Input::get('products.0.name');
```

> - 注意：一些Javascript库，如Backbone可能以JSON的形式发送，你也可以通过 `Input::get` 来访问。

Cookies

laravel框架创建的cookie都是加密并且有认证码的签名，也就是客户端修改会造成cookie的无效。

获取cookie值

```
$value = Cookie::get('name');
```

给响应添加新的cookie

```
$response = Response::make('Hello World');

$response->withCookie(Cookie::make('name', 'value', $minutes));
```

响应前cookie入队列

若想在响应创建前设置cookie，使用 `Cookie::queue()` 方法。这个 `cookie` 会在最终的响应中自动携带。

```
Cookie::queue($name, $value, $minutes);
```

创建Cookie永久有效

```
$cookie = Cookie::forever('name', 'value');
```

原有输入

有时候，你可能想保存输入直到下次请求。比如，在检测验证错误后，需要重新填充表单。

将输入刷入Session

```
Input::flash();
```

将部分输入刷入Session

```
Input::flashOnly('username', 'email');

Input::flashExcept('password');
```

大多时候，可能你只需要将刷入跟跳转的前一个页面关联，那么你可以串联输入刷入到跳转上面。

```
return Redirect::to('form')->withInput();

return Redirect::to('form')->withInput(Input::except('password'));
```

获取原有数据

```
Input::old('username');
```

文件

获取一个上传文件

```
$file = Input::file('photo');
```

判断文件是否上传

```
if (Input::hasFile('photo'))
{
    //
}
```

`Input::file` 方法返回的对象是 `Symfony\Component\HttpFoundation\File\UploadedFile` 的实例，它继承了PHP的 `SplFileInfo` 方法，有文件交互的大量的方法。

判断上传文件是否有效

```
if (Input::file('photo')->isValid())
{
    //
}
```

移动上传文件

```
Input::file('photo')->move($destinationPath);

Input::file('photo')->move($destinationPath, $fileName);
```

获取上传文件的路径

```
$path = Input::file('photo')->getRealPath();
```

获取上传文件的原本名称

```
$name = Input::file('photo')->getClientOriginalName();
```

获取上传文件的扩展名称

```
$extension = Input::file('photo')->getClientOriginalExtension();
```

获取上传文件的大小

```
$size = Input::file('photo')->getSize();
```

获取上传文件的MIME类型

```
$mime = Input::file('photo')->getMimeType();
```

请求信息

`Request` 类提供许多方法来检查HTTP请求和扩展 `Symfony\Component\HttpFoundation\Request` 类，下面是其中一些：

获取请求URI

```
$uri = Request::path();
```

获取请求方式

```
$method = Request::method();

if (Request::isMethod('post'))
{
    //
}
```

判断请求是否匹配指定模式

```
if (Request::is('admin/*'))
{
    //
}
```

获取请求URL

```
$url = Request::url();
```

获取请求URI Segment

```
$segment = Request::segment(1);
```

获取请求Header

```
$value = Request::header('Content-Type');
```

获取$_SERVER数据

```
$value = Request::server('PATH_INFO');
```

判断请求是否是HTTPS

```
if (Request::secure())
{
    //
}
```

判断请求是否是AJAX

```
if (Request::ajax())
{
    //
}
```

判断请求是否是JSON

```
if (Request::isJson())
{
    //
}
```

判断请求是否请求JSON响应数据

```
if (Request::wantsJson())
{
    //
}
```

获取请求响应的数据格式

`Request::format` 方法基于HTTP Accept请求头来返回响应格式：

```
if (Request::format() == 'json')
{
    //
```

```
}
```

视图和响应

基本响应 Response

路由返回字符串

```
Route::get('/', function()
{
    return 'Hello World';
});
```

创建自定义响应

`Response` 的实例继承自 `Symfony\Component\HttpFoundation\Response` 类，提供了很多方法用来构建HTTP响应。

```
$response = Response::make($contents, $statusCode);

$response->header('Content-Type', $value);

return $response;
```

如果你想用 `Response` 类方法，但是返回一个视图作为响应。那么，你可以使用 `Response::view` 方法：

```
return Response::view('hello')->header('Content-Type', $type);
```

响应携带Cookies

```
$cookie = Cookie::make('name', 'value');

return Response::make($content)->withCookie($cookie);
```

跳转 Redirects

返回一个跳转

```
return Redirect::to('user/login');
```

返回一个带有（Flash Data）的跳转

```
return Redirect::to('user/login')->with('message', 'Login Failed');
```

> - 注意：`with`方法会讲数据闪存到session里，你可以通过Session::get方法来获取。

返回一个命名路由的跳转

```
return Redirect::route('login');
```

返回一个携带参数的命名路由的跳转

```
return Redirect::route('profile', array(1));
```

返回一个携带命名参数的命名路由的跳转

```
return Redirect::route('profile', array('user' => 1));
```

返回一个控制器方法的跳转

```
return Redirect::action('HomeController@index');
```

返回一个控制器方法携带参数的跳转

```
return Redirect::action('UserController@profile', array(1));
```

返回一个控制器方法携带命名参数的跳转

```
return Redirect::action('UserController@profile', array('user' => 1));
```

视图 Views

视图包含应用的html代码，使得展现与控制和逻辑分离。视图都放在`app/views`文件夹下。

下面展示一个简答的视图：

```
<!-- View stored in app/views/greeting.php -->

<html>
    <body>
        <h1>Hello, <?php echo $name; ?></h1>
    </body>
</html>
```

这个视图可以如下发送到浏览器端：

```
Route::get('/', function()
{
    return View::make('greeting', array('name' => 'Taylor'));
});
```

传递数据到视图

```
// Using conventional approach
$view = View::make('greeting')->with('name', 'Steve');

// Using Magic Methods
$view = View::make('greeting')->withName('steve');
```

上面代码里的 `$name` 变量将在视图 `greeting.php` 中可用，其值为 `steve` 。你也可以传递一个数组作为 `make` 方法的第二个参数：

```
$view = View::make('greetings', $data);
```

你还可以共享数据给所有的视图：

```
View::share('name', 'Steve');
```

传递一个子视图到一个视图

有时候需要传递一个视图到另一个视图。比如，一个子视图存在在 `app/views/child/view.php` ，那么我们可以像下面传递：

```
$view = View::make('greeting')->nest('child', 'child.view');

$view = View::make('greeting')->nest('child', 'child.view', $data);
```

子视图可以在父视图中被渲染：

```
<html>
    <body>
        <h1>Hello!</h1>
        <?php echo $child; ?>
    </body>
</html>
```

视图组件

视图组件就是在视图渲染时调用回调函数或者类方法。 如果你想在每次渲染视图时，绑定数据到视图，你可以组织这些代码到一个地方。形象地说，视图组件更像是 `视图模型` 或者 `表现者` 。

定义一个视图组件

```
View::composer('profile', function($view)
{
    $view->with('count', User::count());
});
```

现在，每次 `profile` 视图被渲染时， `count` 数据都会被绑定到视图中。

你也可以绑定一个视图组件到多个视图里：

```
View::composer(array('profile','dashboard'), function($view)
{
    $view->with('count', User::count());
});
```

若你想使用类组件而不是匿名函数，也有助于使用 `Ioc` 容器来解析，使用如下：

```
View::composer('profile', 'ProfileComposer');
```

一个类组件可以像下面这样定义：

```
class ProfileComposer {

    public function compose($view)
    {
        $view->with('count', User::count());
    }

}
```

定义多个视图组件

你可以使用 `composer` 方法来一次注册多个视图组件：

```
View::composers(array(
    'AdminComposer' => array('admin.index', 'admin.profile'),
    'UserComposer' => 'user',
    'ProductComposer@create' => 'product'
));
```

> - 注意：类组件没有约束放到哪儿里，你可以随意指定，只要它能被composer.json文件里自动加载使用。

视图创建者

视图创建者类似视图组件，唯一区别就是它会在视图实例化时就会激活。使用 `creator` 方法：

```
View::creator('profile', function($view)
{
    $view->with('count', User::count());
});
```

特殊响应

创建一个json响应

```
return Response::json(array('name' => 'Steve', 'state' => 'CA'));
```

创建一个jsonp响应

```
return Response::json(array('name' => 'Steve', 'state' => 'CA'))->setCallback(Input::get(
```

创建一个文件下载响应

```
return Response::download($pathToFile);

return Response::download($pathToFile, $name, $headers);
```

> - 注意：Symfony HttpFoundation处理文件下载时，限定下载文件名必须是ASCII编码的。

响应Macros

若你想自定义一个响应可以在路由和控制器中多次使用，你可以使用 `Response::macro` 方法：

```
Response::macro('caps', function($value)
{
    return Response::make(strtoupper($value));
});
```

第一个参数作为自定义响应的名称，第二个匿名函数会在 `Response` 类调用这个响应时被调用:

```
return Response::caps('foo');
```

你可以在 `app/start` 文件里选一个定义，又或者你也可以新建一个文件来组件这些 `macros` ，这个文件被 `app/start` 文件里的其中任何一个所引入。

控制器

基本的控制器

相比在路由中定义所有的逻辑，你可能更愿意把这些行为组织到控制器类里。控制器可以把相关的路由逻辑聚集到一个类里，并且可以充分发挥框架的特性，如：自动依赖注入。

控制器通常放在 `app/controllers` 文件夹下，这些文件默认在 `composer.json` 文件里的选项 `classmap` 默认被注册了。当然你也可以放在其他位置，只需要保证 `composer` 可以自动载入这些类。

下面是一个简单的控制器类：

```
class UserController extends BaseController {

    /**
     * Show the profile for the given user.
     */
    public function showProfile($id)
    {
        $user = User::find($id);

        return View::make('user.profile', array('user' => $user));
    }

}
```

所有的控制器类都必须继承 `BaseController` 类，这个类也在 `app/controllers` 里，它被用来存放一些公共的控制逻辑。 `BaseControllers` 类又继承了框架的 `Controllers` 类。 现在，你可以如下路由你的控制器方法：

```
Route::get('user/{id}', 'UserController@showProfile');
```

若你想用命名空间来组织和嵌套你的控制器，定义路由时需要使用完整的类命名：

```
Route::get('foo', 'Namespace\FooController@method');
```

> - 注意：因为我们是用 `composer` 来自动加载php类，控制器可以在任何地方，只要 `composer` 可以加载到。

你也可以给控制器路由指定名称：

```
Route::get('foo', array('uses' => 'FooController@method',
                                        'as' => 'name'));
```

通过 `URL::action` 方法或者action帮助函数可以获取控制器方法的url:

```
$url = URL::action('FooController@method');

$url = action('FooController@method');
```

通过 `Route::currentRouteAction` 来获取当前控制器方法的名称：

```
$action = Route::currentRouteAction();
```

控制器过滤器

过滤器同正匹配路由一样，指定到相对应的路由器：

```
Route::get('profile', array('before' => 'auth',
            'uses' => 'UserController@showProfile'));
```

当然，你也可以在控制器内部来指定过滤器：

```
class UserController extends BaseController {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->beforeFilter('auth', array('except' => 'getLogin'));

        $this->beforeFilter('csrf', array('on' => 'post'));

        $this->afterFilter('log', array('only' =>
                            array('fooAction', 'barAction')));
    }

}
```

你也可以通过匿名函数来指定过滤器：

```
class UserController extends BaseController {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->beforeFilter(function()
        {
            //
        });
    }
```

```
}
```

你也可以使用当前控制器的其他方法作为控制器，使用 `@` 前缀来定义：

```
class UserController extends BaseController {

    /**
     * Instantiate a new UserController instance.
     */
    public function __construct()
    {
        $this->beforeFilter('@filterRequests');
    }

    /**
     * Filter the incoming requests.
     */
    public function filterRequests($route, $request)
    {
        //
    }

}
```

隐式控制器

laravel可以很容易的用一个路由器来处理一个控制器的所有方法，使用 `Route::controller` 如下：

```
Route::controller('users', 'UserController');
```

接收两个参数。第一个是控制器处理的base URI，第二个是控制器类名。接下里，我们给控制器每个方法添加 `HTTP 方式` 前缀：

```
class UserController extends BaseController {

    public function getIndex()
    {
        //
    }

    public function postProfile()
    {
        //
    }

    public function anyLogin()
    {
        //
    }

}
```

上例 `index` 方法对应的root URI 是 `users` 。

如果你的控制器方法有多个单词，你可以在url地址里通过 `短斜线` 来使用。比如URI为 `users/admin-profile` 对应控制器 `UserController` 的方法为：

```
public function getAdminProfile() {}
```

RESTful 资源控制器

资源控制器可以很容易的创建关于 `resources` 的Restful控制器。比如，你可能需要创建一个控制器来处理应用中的 `图片` 。通过 `Artisan CLI` 和 `Route::resource` 可以构建这样一个控制器：

```
php artisan controller:make PhotoController
```

接下里，给这个控制器注册一个资源模式的路由：

```
Route::resource('photo', 'PhotoController');
```

这个路由定义创建了一系列对照片RESTful操作的路由。同样滴，生成的控制器也包含了一些列的方法，并有相对应的URI和请求方式。

资源控制器处理的方法：

Verb	Path	Action	Route Name
GET	/resource	index	resource.index
GET	/resource/create	create	resource.create
POST	/resource	store	resource.store
GET	/resource/{resource}	show	resource.show
GET	/resource/{resource}/edit	edit	resource.edit
PUT/PATCH	/resource/{resource}	update	resource.update
DELETE	/resource/{resource}	destroy	resource.destroy

如果你只需要其中一部分方法的话，可以这样:

```
php artisan controller:make PhotoController --only=index,show

php artisan controller:make PhotoController --except=index
```

也可以给路由指定一部分的方法：

```
Route::resource('photo', 'PhotoController',
                array('only' => array('index', 'show')));
```

```
Route::resource('photo', 'PhotoController',
                array('except' => array('create', 'store', 'update', 'destroy')));
```

默认地，资源控制器都有一个路由名称。你也可以通过一个下标为 `names` 的数组来覆盖默认名称：

```
Route::resource('photo', 'PhotoController',
                array('names' => array('create' => 'photo.build')));
```

处理嵌套的资源控制器：

路由定义中，使用 `.` 符号来使用嵌套：

```
Route::resource('photos.comments', 'PhotoCommentController');
```

这个路由会注册一个嵌套的资源，你可以通过urls类似：`photos/{photoResource}/comments/{commentResource}` 来访问。

```
class PhotoCommentController extends BaseController {

    public function show($photoId, $commentId)
    {
        //
    }

}
```

给资源控制器添加其他路由

若需要给你的资源控制器添加其他路由，除了默认资源路由外，你需要在调用 `Route::resource` 之前来定义这些路由：

```
Route::get('photos/popular');
Route::resource('photos', 'PhotoController');
```

处理缺少方法

在使用 `Route::controller` 时，可以在控制器内部定义个捕获方,在没有匹配到控制器的方法时自动调用。这个方法名称为 `misssingMethod` ，并且接收请求的方法和参数数组。

定义如下：

```
public function missingMethod($parameters = array())
{
    //
}
```

```
```

在使用资源控制器时，你可能还需要在该控制器上定义 `__call` 魔术方法来调用 `missingMethod` 。

查询生成器

介绍

查询生成器提供了方便流畅的接口来创建和执行queries。它可以执行你应用的大多数的数据库操作，并且适配于所有支持的数据库。

> - 注意：Laravel的查询生气使用PDO的参数绑定来保护您的应用防止sql注入，绑定的参数不需要额外进行过滤处理。

Selects

获取表的所有记录

```
$users = DB::table('users')->get();

foreach ($users as $user)
{
    var_dump($user->name);
}
```

获取表的一条记录

```
$user = DB::table('users')->where('name', 'John')->first();

var_dump($user->name);
```

获取表的某行的一列数据

```
$name = DB::table('users')->where('name', 'John')->pluck('name');
```

获取表的某列数据的列表

```
$roles = DB::table('roles')->lists('title');
```

这个方法会返回 `role title` 的数组。你可以给返回结果指定一个自定义的key：

```
$roles = DB::table('roles')->lists('title', 'name');
```

指定查询语句

```
$users = DB::table('users')->select('name', 'email')->get();
```

```
$users = DB::table('users')->distinct()->get();

$users = DB::table('users')->select('name as user_name')->get();
```

当前Query语句添加查询语句

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

使用where条件

```
$users = DB::table('users')->where('votes', '>', 100)->get();
```

or语法

```
$users = DB::table('users')
                ->where('votes', '>', 100)
                ->orWhere('name', 'John')
                ->get();
```

where: between

```
$users = DB::table('users')
                ->whereBetween('votes', array(1, 100))->get();
```

where: not between

```
$users = DB::table('users')
                ->whereNotBetween('votes', array(1, 100))->get();
```

where:in

```
$users = DB::table('users')
                ->whereIn('id', array(1, 2, 3))->get();

$users = DB::table('users')
                ->whereNotIn('id', array(1, 2, 3))->get();
```

where:null

```
$users = DB::table('users')
                    ->whereNull('updated_at')->get();
```

order by, group by 和 having

```
$users = DB::table('users')
                    ->orderBy('name', 'desc')
                    ->groupBy('count')
                    ->having('count', '>', 100)
                    ->get();
```

offset & Limit

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

联接（Joins）

查询生成器也可以写join语法，看下面的例子：

基本的join语法

```
DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.id', 'contacts.phone', 'orders.price')
            ->get();
```

left join语法

```
DB::table('users')
        ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
        ->get();
```

你可以使用更高级的join语句：

```
DB::table('users')
        ->jon('contacts', function($join)
        {
            $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
        })
        ->get();
```

若你喜欢 `where` 风格来处理joins,你可以在join上使用 `where` 和 `orWhere` 方法。这些方法不是比较两列，而是某列与值的比较：

```
DB::table('users')
        ->join('contacts', function($join)
```

```
        {
            $join->on('users.id', '=', 'contacts.user_id')
                    ->where('contacts.user_id', '>', 5);
        })
        ->get();
```

where的高级用法

参数化映射

当你需要构建高级的where语句时，如： `where exists` 或者嵌套的参数组，查询生成器也可以处理它：

```
DB::table('users')
        ->where('name', '=', 'John')
        ->orWhere(function($query)
        {
            $query->where('votes', '>', 100)
                  ->where('title', '<>', 'Admin')
        })
        ->get();
```

上面的query会产生如下sql:

```
select * from users where name = 'John' or (votes > 100 and title <> 'Admin')
```

Exits语法

```
DB::table('users')
            ->whereExists(function($query)
            {
                $query->select(DB::raw(1))
                      ->from('orders')
                      ->whereRaw('orders.user_id = users.id');
            })
            ->get();
```

上面的query会产生如下sql:

```
select * from users
where exists (
    select 1 from orders where orders.user_id = users.id
)
```

聚合

查询生成器也有聚合的方法，如： `count` / `max` / `min` / `avg` / `sum` .

使用聚合方法：

```
$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');

$price = DB::table('orders')->min('price');

$price = DB::table('orders')->avg('price');

$total = DB::table('orders')->sum('price');
```

原生(Raw)表达式

有时候你需要用原生表达式，这些表达式会作为字串来注入query，所以你需要特别小心sql注入。使用 `DB::raw` 方法：

使用原生表达式

```
$users = DB::table('users')
             ->select(DB::raw('count(*) as user_count, status'))
             ->where('status', '<>', 1)
             ->groupBy('status')
             ->get();
```

插入（Inserts）

插入一条记录

```
DB::table('users')->insert(
    array('email' => 'john@example.com', 'votes' => 0);
);
```

插入一条记录并获取自增id

```
$id = DB::table('users')->insertGetId(
    array('email' => 'john@example.com', 'votes' => 0);
);
```

> - 注意若使用的是PostgreSQL，自增列的名称限定为id，其它数据库不限定名称。

插入多条记录

```
DB::table('users')->insert(
    array('email' => 'taylor@example.com', 'votes' => 0),
    array('email' => 'daylee@example.com', 'votes' => 0)
);
```

更新（Updates）

更新一条记录

```
DB::table('users')
    ->where('id', 1)
    ->update(array('votes' => 1));
```

某列自增或自减指定值

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

你也可以同时指定其它列的更新

```
DB::table('users')->increment('votes', 1, array('name' => 'John'));
```

删除（Deletes）

删除表指定记录

```
DB::table('users')->where('votes', '<', 100)->delete();
```

删除表所有记录

```
DB::table('users')->delete();
```

truncate表

```
DB::table('users')->truncate();
```

联合（Unions）

查询生成器提供了一个快捷的方式来 `union` 来个query:

```
$first = DB::table('users')->whereNull('first_name');

$users = DB::table('users')->whereNull('last_name')->union($first)->get();
```

`unionAll` 方法也可以使用，跟 `union` 使用方法一样。

悲观锁

查询生成器提供一些功能帮助你处理SELECT语句。

若需要执行 `shared lock` ，你需要在query上使用 `sharedLock` 方法：

```
DB::table('users')->where('votes', '>', 100)->sharedLock()->get();
```

为了给SELECT语句添加更新锁，你需要在query上使用 `lockForUpdate` 方法：

```
DB::table('users')->where('votes', '>', 100)->lockForUpdate()->get();
```

缓存Queries

使用 `remember` 方法可以很容易的缓存query的结果

```
$users = DB::table('users')->remember(10)->get();
```

上例中，query的结果会被缓存10分钟。在缓存有效期内，这个query不会查询数据库，它会从你应用指定的缓存驱动器上加载数据。

如果你在使用支持的缓存驱动器，你可以给缓存添加标签：

```
$users = DB::table('users')->cacheTags(array('people', 'authors'))->remember(10)->get();
```

基本用法

配置

laravel配置数据库、执行 `queries` 非常简单。配置文件在 `app/config/database.php` 里，你可以定义定义所有的数据库连接及默认地数据库连接。范例中包含了了所有支持的数据库连接。

目前之前的数据库有：MySQL/Postgres/SQLite/SQL Searver。

读/写连接

有时候我们需要在 `SELECT` 语句时选择一个数据库连接，而在 `INSERT` , `UPDATE` , `DELETE` 语句时选择其他的数据库连接，laravel可以非常简单处理，不管你用的是 原生queries，查询生成器还是Eloquent ORM。

如何配置读/写连接，看下面的例子：

```
'mysql' => array(
    'read' => array(
        'host' => '192.168.1.1',
    ),
    'write' => array(
        'host' => '196.168.1.2'
    ),
    'driver'    => 'mysql',
    'database'  => 'database',
    'username'  => 'root',
    'password'  => '',
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
),
```

注意我们添加了两个keys： `read` 和 `write` 。它们共享mysql里所有的公共配置，需要自定义的选项可以添加到 `read` 和 `write` 对应的数组里。

执行Queries

数据库连接配置成功后，你就可以如下使用 `DB` 类来执行queries。

执行Select查询

```
$results = DB::select('select * from users where id = ?', array(1));
```

select方法会返回结果集数组。

执行Insert语句

```
DB::insert('insert into users (id, name) values (?, ?)', array(1, 'Dayle'));
```

执行Update语句

```
DB::update('update users set votes = 100 where name = ?', array('John'));
```

执行Delete语句

```
DB::delete('delete from users');
```

> - 注意：update和delete语句会返回操作的影响行数。

执行一般语句

```
DB::statement('drop table users');
```

监听所有Query的事件

使用 `DB::listen` 方法：

```
DB::listen(function($sql, $bindings, $time)
{
    //
});
```

事务

为执行一系列数据库事务的操作，你需要使用 `transaction` 方法：

```
DB::transaction(function()
{
    DB::table('users')->update(array('votes' => 1));

    DB::table('posts')->delete();
});
```

> - 注意： `transaction` 闭包里抛出的异常会使得这个事务在后台自动回滚。

若需手动启动一个事务： `beginTransaction`

```
DB::beginTransaction();
```

回滚： `rollback`

```
DB::rollback();
```

提交：`commit`

```
DB::commit();
```

获取数据库连接

如果你有多个连接时，可以通过 `DB::connection` 方法来获取：

```
$users = DB::connection('foo')->select(...);
```

也可以获取原生部署的pdo实例：

```
$pdo = DB::connection->getPdo();
```

若数据库连接超过了pdo实例的 `max_connections` 限制，那么可以使用 `disconnect` 方法来断开：

```
DB::disconnect('foo');
```

Query日志

默认地，laravel会讲当前请求的执行的所有queries放在内存里。但是，当我们的插入大量的数据时，可能会占用过多的内存。那么，你可以使用 `disableQueryLog` 来禁用它：

```
DB::connection()->disableQueryLog();
```

若想获取执行的queries，使用 `getQueryLog` 方法：

```
$queries = DB::getQueryLog();
```

Eloquent ORM

介绍

Laravel包含的Eloquent ORM提供了处理数据库的一套优美、简易的ActiveRecord实现。每个数据表有一个与之对应的 `模型` 来操作数据表。

开始之前，确认你配置好了 `app/config/database.php` 中的数据库连接。

基本用法

首先，让我们创建一个Eloquent模型，默认在 `app/models` 文件夹下，你也可以在 `composer.json` 文件里随意指定它的位置，只要确保被自动加载。

定义Eloquent模型

```
class User extends Eloquent{}
```

注意，我们并没有特意指定这个模型使用的数据表。其实这个类名的小写和复数就是默认的数据表，除非你特意指定。上面 `User` 模型对应的数据表为 `users` ，若需要自定义表名，在这个模型类中添加 `protected` 的属性 `$table` 即可：

```
class User extends Eloquent{
    protected $table = 'my_users';
}
```

> - 注意：Eloquent默认假设每个表有一个名为 `id` 的主键，你可以定义 `primaryKey` 属性来覆盖它。同样地，使用模型时用到的数据库连接，你也可以定义 `connection` 属性来覆盖它。

一旦模型被定义，你就可以获取和创建数据表记录。注意，默认你需要给数据表指定 `updated_at` 和 `created_at` 列，如果你希望自动更新，设置 `$timestamps` 属性为 `false` ;

获取所有数据

```
$users = User::all();
```

通过主键来获取一条记录

```
$user = User::find(1);

var_dump($user->name);
```

> - 注意：所有查询生成器的方法，Eloqunet模型也可以使用。

通过主键获取一条数据，失败抛出异常

有时候你希望模型未找到时抛出一个异常，这样就可以用 `App:error` 处理器来捕获它，并展示404页面：

```
$model = User::findOrFail(1);

$model = User::where('votes', '>', 100)->firstOrFail();
```

注册错误处理器，监听 `ModelNotFoundException`：

```
use Illuminate\Database\Eloquent\ModelNotFoundException;

App::error(function(ModelNotFoundException $e)
{
    return Response::make('Not Found', 404);
});
```

query使用Eloquent模型

```
$users = User::where('votes', '>', 100)->take(10)->get();

foreach ($users as $user)
{
    var_dump($user->name);
}
```

Eloquent聚合

你也可以使用查询生成器的聚合方法：

```
$count = User::where('votes', '>', 100)->count();
```

如果你不能通过查询生成器来生成query，你可以使用 `whereRaw`：

```
$users = User::whereRaw('age > ? and votes = 100', array(25))->get();
```

拆分查询

若需要处理大量（数以千计）的Eloquent记录，使用 `chunk` 方法可以防止吃掉太多的内存：

```
User::chunk(200, function($users)
{
    foreach ($users as $user)
        {
            //
        }
```

```
});
```

第一个参数200即时你每个 `chunk` 处理的记录数。闭包回调参数 `$users` 是数据库查询的记录，查询完成后自动调用。

指定某个query的数据库链接

使用 `on` 方法可以指定某个数据库连接以查询这个query

```
$user = User::on('connection-name')->find(1);
```

批量赋值（Mass Assignment）

创建模型后，我们可以传递一个属性数组给模型构造器。这些属性数组通过 `Mass Assignment` 赋值给这个模型。虽然非常方便，但是，盲目的赋值可能会有安全的隐患。如果用户可以随意的传递输入，就可以随意更改模型的属性。出于这个考虑，所有的模型会防范批量赋值。

开始前，需要给模型设置 `fillable` 和 `guarded` 属性：

给模型定义Fillable属性

`fillable` 属性指定了批量赋值哪儿些属性。可以在类或者实例类设置：

```
class User extends Eloquent {
    protected $fillable = array('first_name', 'last_name', 'email');
}
```

如上，只有列出的三个属性可以进行批量赋值。

给模型定义Guarded属性

与 `fillable` 相反的是 `guarded`，类似于黑名单一样。

```
class User extends Eloquent{
    protected $guarded = array('id', 'password');
}
```

> - 注意：当使用 `guarded` 时，你不可以使用 `Input::get()` 或者任何其他用户输入的原生数组给 `save` 和 `update` 方法，因为没在 `guarded` 中的字段可能被更新。

阻止批量赋值里的所有属性

上例中， `id` 和 `password` 属性不能批量赋值，而其他的可以。若想阻止所有属性：

```
protected $guarded = array('*');
```

插入、更新、删除

添加一条记录

```
$user = new User;

$user->name = 'John';

$user->save();
```

> - 注意：通常情况下，Eloquent模型有自增键。如果你需要指定自己的，在模型中设置 `incrementing` 属性为 `false` 。

你可以使用 `create` 方法，使用一行代码就可以创建一个新的模型。 `create` 方法会返回插入模型的实例。当然了，你必须指定属性 `fillable` 或者 `guarded` ，因为所有的 `Eloquent` 模型都防范批量赋值。

使用自增id保存或者新建模型实例后，你可以通过 `id` 属性来获取它的值：

```
$insertedId = $user->id;
```

给模型设置 `Guarded` 属性

```
class User extends Eloquent{
    protected $guarded = array('id', 'account_id');
}
```

使用模型的 `create` 方法

```
// Create a new user in the database...
$user = User::create(array('name' => 'John'));

// Retrieve the user by the attributes, or create it if it doesn't exist...
$user = User::firstOrCreate(array('name' => 'John'));

// Retrieve the user by the attributes, or instantiate a new instance...
$user = User::firstOrNew(array('name' => 'John'));
```

获取模型后更新

你可以先获取一个模型实例，改变它的属性值，然后用 `save` 方法来更新：

```
$user = User::find(1);

$user->email = 'john@foo.com';

$user->save();
```

保存模型及关系

若需要保存一个模型实例及所有关系，可以使用 `push` 方法：

```
$user->push();
```

你也可以以query的方式来执行udpate

```
$affectedRows = User::where('votes', '>', 100)->update(array('status' => 2));
```

> - 注意：当你使用Eloquent的查询生成器时，不会激活任何的事件

删除存在的模型实例

调用 `delete` 方法：

```
$user = User::find(1);

$user->delete();
```

通过主键值来删除模型实例

```
User::destroy(1);

User::destroy(array(1, 2, 3));

User::destroy(1, 2, 3);
```

当然你可以在一系列的模型实例上执行一个删除query：

```
$affectedRows = User::where('votes', '>', 100)->delete();
```

只更新模型的**timestamps**字段

若纸箱更新模型的 `timestamps` 字段，使用 `touch` 方法：

```
$user->touch();
```

软删除

当执行软删除时，并不会真正的从数据库删除数据，而是在记录的 `deleted_at` 时间戳进行设置。使用它需要在模型里应用 `SoftDeletingTrait` ：

```
user Illuminate\Database\Eloquent\SoftDeletingTrait

class User extends Eloquent{

    use SoftDeletingTrait;

    protected $dates = ['deleted_at'];
}
```

数据表添加 `deleted_at` 字段后，就可以在数据库迁移时用 `softDeletes` 方法：

```
$table->softDeletes();
```

现在在模型上使用 `delete` 方法时，对应字段 `deleted_at` 值会设置为当前时间戳。当查询一个使用了软删除的模型时，“软删除”的数据不会包含到结果中。

强制软删除数据显示到结果集中

若想强制显示软删除数据，query时使用 `withTrashed` 方法：

```
$user = User::withTranshed()->where('account_id', 1)->get();
```

`withTranshed` 方法在关联时依然有效：

```
$user->posts()->withTrashed()->get();
```